CSCI2467: Systems Programming Concepts

Class activity: `bash` shell literacy

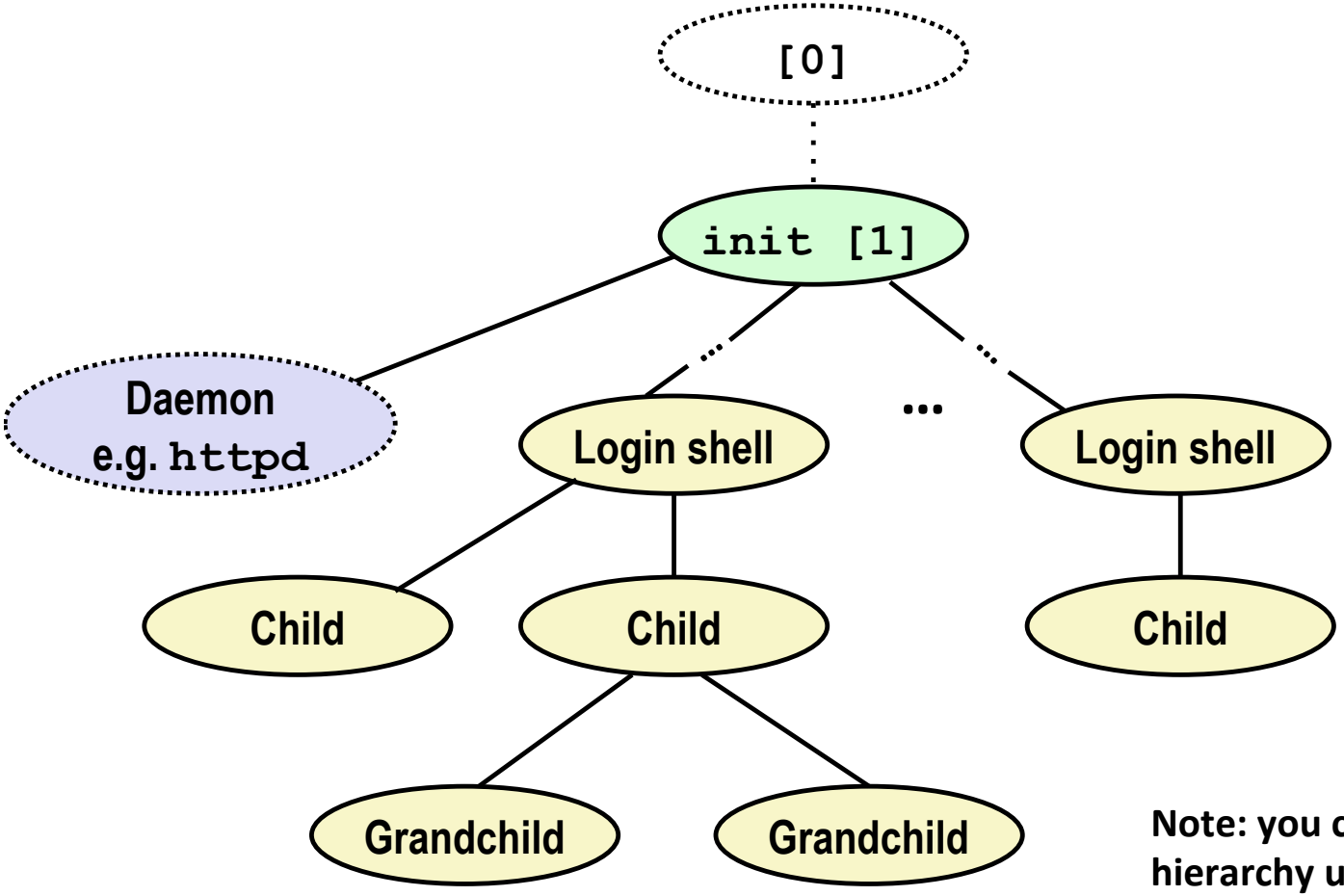
Spring 2020

THE UNIVERSITY of NEW ORLEANS
DEPARTMENT OF COMPUTER SCIENCE

Today

1. Shells
   - History
   - Usage
   - Scripts vs. Programs
2. Bash shell: practical uses for your systems skills
   - Computing your score so far
   - More common tools
3. Activity
   - Make a shell script

Linux process hierarchy



Note: you can view the hierarchy using the Linux `pstree` **command**

Shell

- A shell is an application program that runs programs on behalf of the user.

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

  (`vi`, Sun Microsystems, Java language)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

  (`vi`, Sun Microsystems, Java language)
- `bash` GNU "Bourne-Again" Shell (1989)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

  (`vi`, Sun Microsystems, Java language)
- `bash` GNU "Bourne-Again" Shell (1989)

  default Linux shell (1991), default macOS shell (since 10.3, 2003)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

  (`vi`, Sun Microsystems, Java language)
- `bash` GNU "Bourne-Again" Shell (1989)

  default Linux shell (1991), default macOS shell (since 10.3, 2003)

  Windows 10 (Anniversary update, 2016)

Shell

- A shell is an application program that runs programs on behalf of the user.

- `sh` Orig. Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- `csh` / `tcsh` BSD Unix C shell (1978 / 1981)

  1978: UC Berkeley grad student Bill Joy

  (`vi`, Sun Microsystems, Java language)
- `bash` GNU "Bourne-Again" Shell (1989)

  default Linux shell (1991), default macOS shell (since 10.3, 2003)

  Windows 10 (Anniversary update, 2016)
- `tsh` "tiny" shell (you, 2020)

bash is used in many ways

Command processor, or programming language?

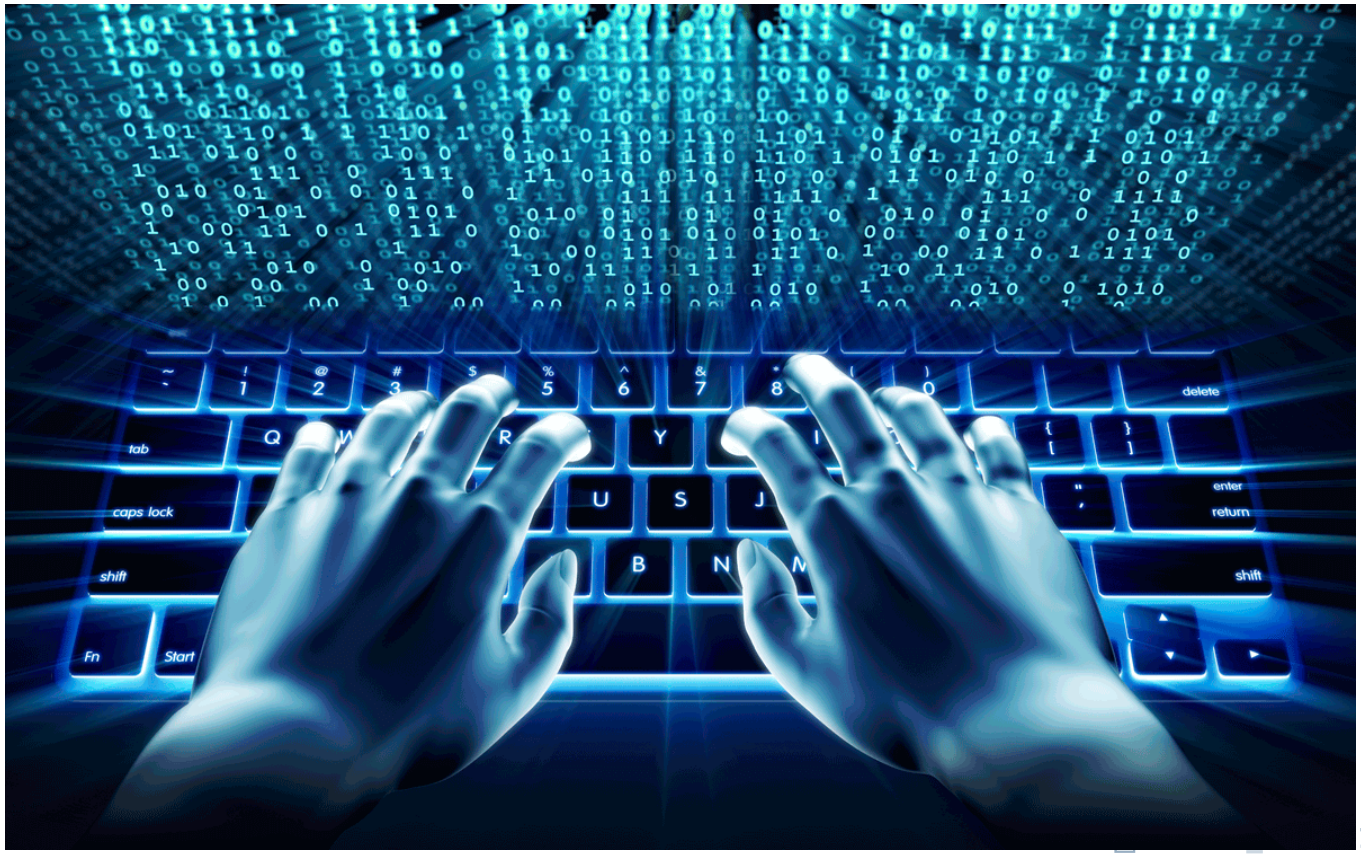
bash is used in many ways

Command processor, or programming language?

Both!

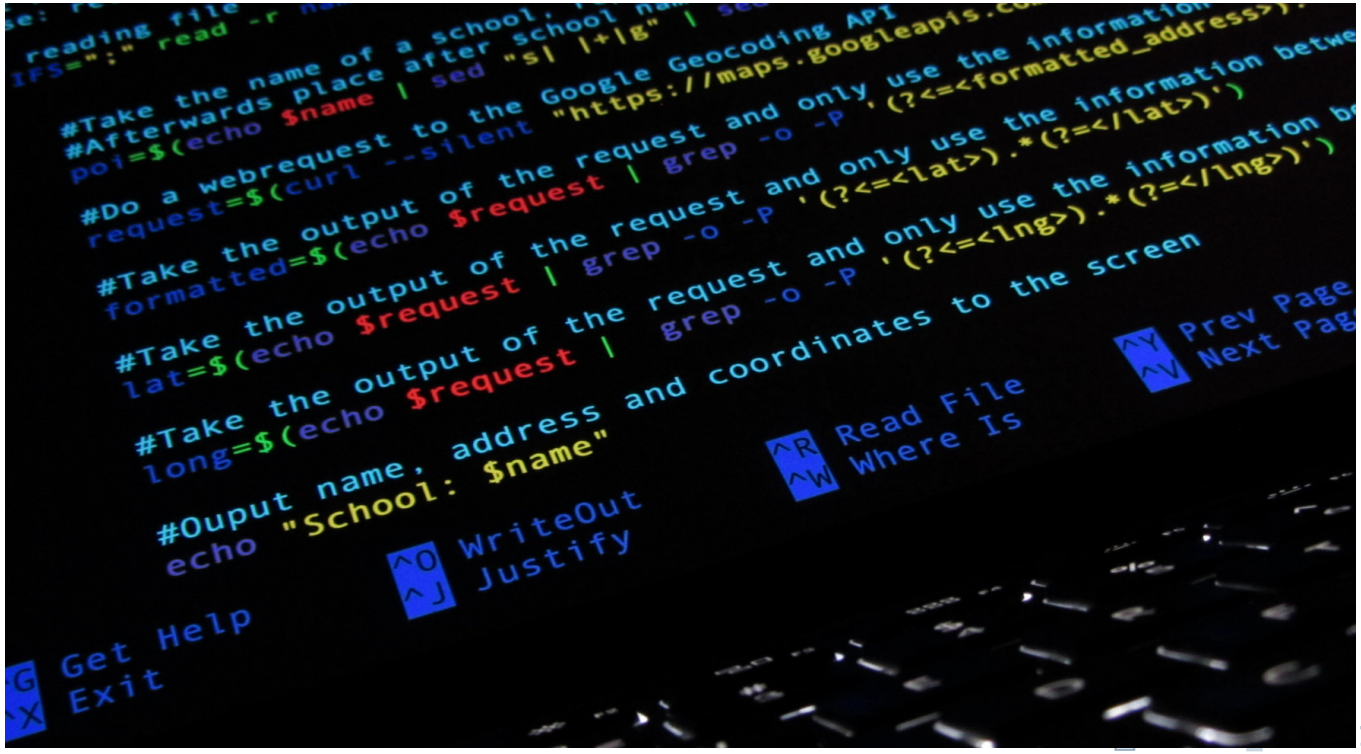
Interactive vs. scripting

- Interactive case: human types commands at a prompt
- You've been using `bash` this way.
- Your `tsh` will work this way.



Interactive vs. scripting

- Shell script: any commands you could type at the shell prompt could also go into a file
  - `bash` then *interprets* that file as a sequence of commands
  - thie file is typically called a `script` (denoted by `.sh` filename extension)



Interactive vs. scripting

`bash` as Command Processor

- Interactive case: human types commands at a prompt
  - You've been using `bash` this way.
  - Your `tsh` works this way.

`bash` as Programming language

- Shell script: any commands you could type at the shell prompt could also go into a file
  - `bash` then *interprets* that file as a sequence of commands
  - thie file is typically called a `script`

    (denoted by `.sh` filename extension)

Script? Program? What's the difference?

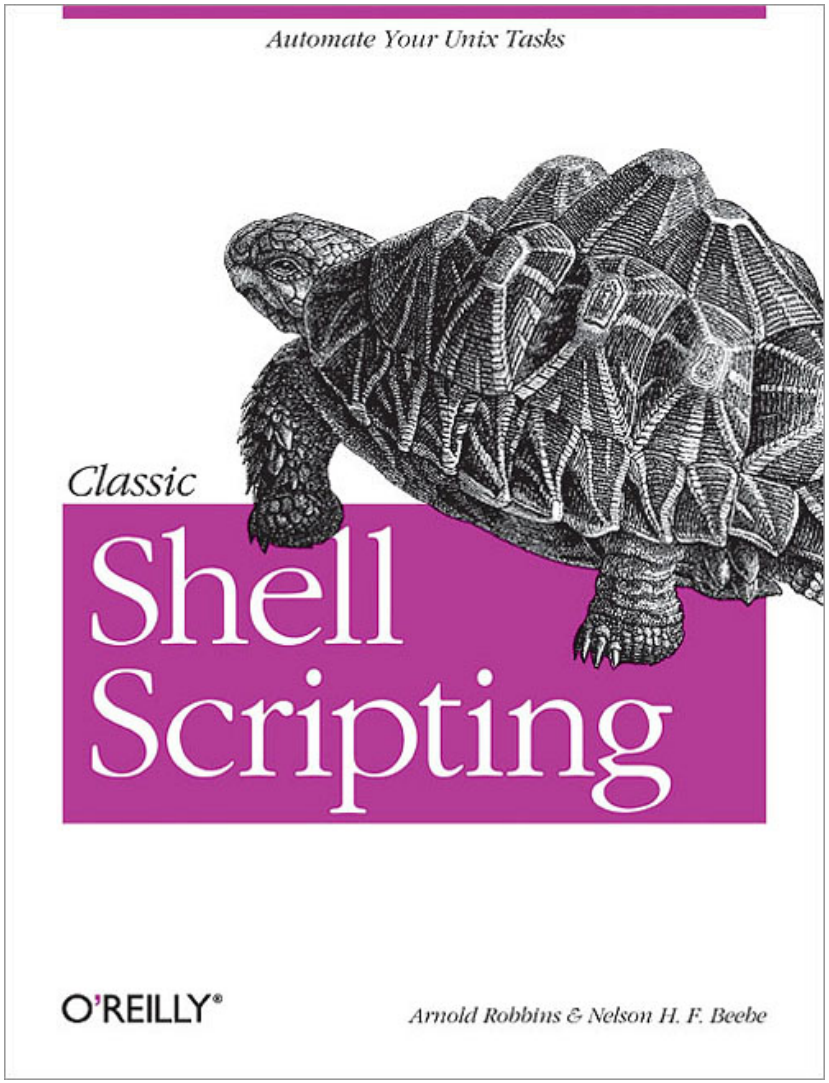
- scripts are text files which are *interpreted*
- Scripting languages: bash, perl, python (many more)
- An interpreter reads script text every time it runs and then performs commands

Script? Program? What's the difference?

- scripts are text files which are *interpreted*
  - Scripting languages: bash, perl, python (many more)
  - An interpreter reads script text every time it runs and then performs commands

- Traditional programming languages are *compiled*
  - A compiler converts human-readable source code to machine-level instructions (may be called binary or executable)
  - These machine-level instructions are the focus of Chapter 3

Shell scripting



Today

1. Shells
   - History
   - Usage
   - Scripts vs. Programs
2. Bash shell: practical uses for your systems skills
   - Computing your score so far
   - More common tools
3. Activity
   - Make a shell script

Your point total

- As usual, log on to terminal in Math 209/212 or via ssh to `systems-lab.cs.uno.edu`

Your point total

- Change to your 2467 directory:

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

```
$ echo 40 > SCORE.lab0
```

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

```
$ echo 40 > SCORE.lab0
```

- Can view all with:

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

```
$ echo 40 > SCORE.lab0
```

- Can view all with:

```
$ cat SCORE.*
```

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

```
$ echo 40 > SCORE.lab0
```

- Can view all with:

```
$ cat SCORE.*
```

- ... and add up with:

Your point total

- Change to your 2467 directory:

```
$ cd 2467
```

- Create files called:

```
SCORE.midterm SCORE.lab0 SCORE.lab1 SCORE.lab2
```

- Could use an editor, but it is faster to do this:

```
$ echo 40 > SCORE.lab0
```

- Can view all with:

```
$ cat SCORE.*
```

- ... and add up with:

```
$ cat SCORE.* | /home/CSCI2467/sum.sh
```

What did we just do?

No felines were harmed

- `cat` command concatenates files

  output contents of each file given in order

  if only one filename given (singleton), then simply output contents of that file

What did we just do?

No felines were harmed

- `cat` command concatenates files

  output contents of each file given in order

  if only one filename given (singleton), then simply output contents of that file

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  `SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm`

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  `SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm`
- `bash` does this, so the `cat` program thinks you typed:

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  ```
  SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

- `bash` does this, so the `cat` program thinks you typed:

  ```
  cat SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  ```
  SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

- `bash` does this, so the `cat` program thinks you typed:

  ```
  cat SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

- wildcards won't work in `tsh` or `tshref`

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  `SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm`
- `bash` does this, so the `cat` program thinks you typed:

  `cat SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm`

- wildcards won't work in `tsh` or `tshref`
- so in `tsh` you would see this:

What did we just do?

wildcards, or "globbing"

- Bash looked in the current directory for files that match the pattern: `SCORE.*`
- `*` is a *wildcard*: matches anything (zero or more characters)
- Therefore `SCORE.*` matches any file which begins with `SCORE.`
- As in, all of your scores:

  ```
  SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

- `bash` does this, so the `cat` program thinks you typed:

  ```
  cat SCORE.lab0 SCORE.lab1 SCORE.lab2 SCORE.midterm
  ```

- wildcards won't work in `tsh` or `tshref`
- so in `tsh` you would see this:

  ```
  /bin/cat:  SCORE.*:  No such file or directory
  ```

What did we just do?

more on wildcards

- Could also match a single character:

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

  - Would match `SCORE.lab9` but *not* `SCORE.lab10`

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

  - Would match `SCORE.lab9` but *not* `SCORE.lab10`

- Or, match only certain characters:

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

  - Would match `SCORE.lab9` but *not* `SCORE.lab10`

- Or, match only certain characters:

  `cat SCORE.lab[01]`

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

  - Would match `SCORE.lab9` but *not* `SCORE.lab10`

- Or, match only certain characters:

  `cat SCORE.lab[01]`

  - matches only `SCORE.lab0` and `SCORE.lab1`

What did we just do?

more on wildcards

- Could also match a single character:

  `cat SCORE.lab?`

  - Would match `SCORE.lab9` but *not* `SCORE.lab10`

- Or, match only certain characters:

  `cat SCORE.lab[01]`

  - matches only `SCORE.lab0` and `SCORE.lab1` (not `SCORE.lab2`) or any others

What did we just do?

more on wildcards

- Try creating a SCORE.lab10 file with:

  `echo 100 > SCORE.lab10`

  - remember *output redirection?*
- Then compare:

  `cat SCORE.lab?`

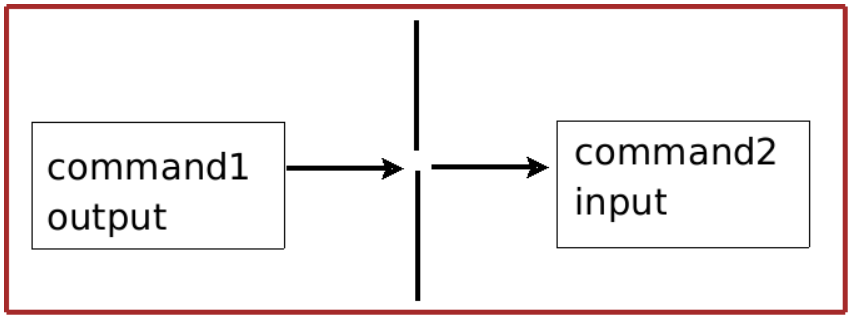
  `cat SCORE.lab??`

  `cat SCORE.lab*`
- Delete `SCORE.lab10` when finished:

  `rm SCORE.lab10`
- Be **very** careful using `*` with `rm` !

What did we just do?

Pipes



- | is called a "pipe"
  - `cmd1` output normally goes to the screen

    ... but instead becomes `cmd2` input (another type of redirection)
  - Then `cmd2` output goes to screen

    ... but *could* be written to a file:

    ```
    cmd1 | cmd2 > outputfile
    ```
- *Many* more examples on Bash redirections cheat sheet

What did we just do?

Shell script example

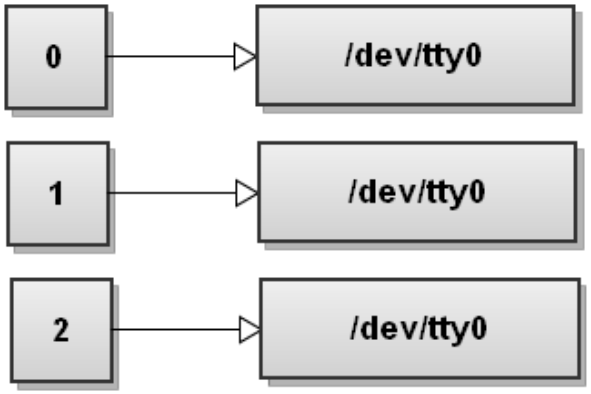
```
#!/bin/bash

TOTAL=0
while read val; do
        TOTAL=$(($TOTAL+$val))
done
echo $TOTAL
```

- `sum.sh` is a *shell script* (text file)
- commands interpreted by program `/bin/bash`
- would also work if typed in at the prompt:

```
TOTAL=0;
while read val; do TOTAL=$(($TOTAL+$val)); done;
echo $TOTAL
```

What did we just do?

stdin

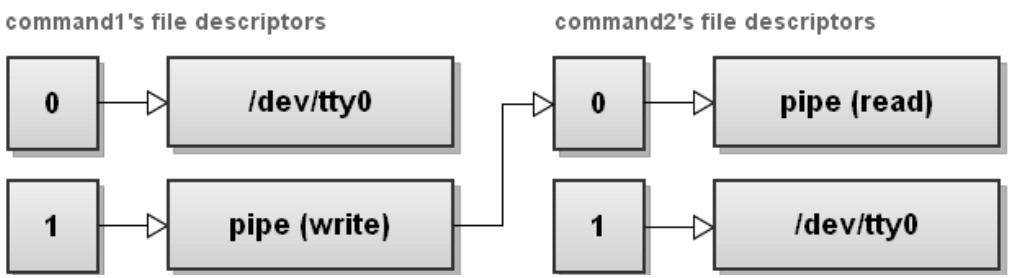


- `sum.sh` run by itself takes input from keyboard (each number on its own line, end with `CTRL-D`)
- file descriptors 0, 1 and 2 commonly referred to as: `stdin, stdout, stderr`

Images from `http://www.catonmat.net/blog/bash-one-liners-explained-part-three/`

What did we just do?

stdin



- Thanks to the pipe we used, instead the input to the sum program came from those `SCORE.*` files

  (actually `sum.sh` didn't read files, it read the output of `cat`)
- keeps `sum.sh` program very simple
- Also works with C programs (file descriptors 0 and 1, `open()` and `dup2()`)

Images from `http://www.catonmat.net/blog/bash-one-liners-explained-part-three/`

grep: print lines matching a pattern

Make yourself a copy of this example file:

```
cp /home/CSCI2467/labs/bash/lab0-comments .
```

Then run:

```
grep part lab0-comments
```

- strange name, used as noun or verb
- simple and very useful
- example above: find any line containing "part" in file `lab0-comments` (contains several lines)
- case sensitive! try with `Part` instead
- (try with option `-i` for insensitive)
- could write result to a file with:

  ```
  grep Part lab0-comments > mylab0parts
  ```

Pipes make grep even more useful

Pipes make grep even more useful

```
#Take the output of the request and only use the information
long=$(echo $request | grep -o -P '(?<=<lat>).*(?=</lat
#Ouput name, address and coordinates to the screen
echo "School: $name"
echo "Address: $formatted"
echo "Coordinates: $lat , $long"
#Go to sleep for 5 seconds (do
sleep 5
```

Today

1. Shells
   - History
   - Usage
   - Scripts vs. Programs
2. Bash shell: practical uses for your systems skills
   - Computing your score so far
   - More common tools
3. Activity
   - Make a shell script

Extra credit opportunity

- Go to AutoLab
- look for extra credit **bash scripting**
- the *writeup* is these slides
  (at the end of the PDF files)
- the *handout* is called `grep.tar`

Extra credit opportunity

- Go to AutoLab
- look for extra credit **bash scripting**
- the *writeup* is these slides

  (at the end of the PDF files)
- the *handout* is called `grep.tar`

  also can be found at:

  `/home/CSCI2467/labs/misc/grep.tar`

Extra credit opportunity

- Add up to 5 points to your point total
- Create a shell script called `secretfinder.sh` which:
  - create a directory called `secretfiles` and change into it
  - extract files from `grep.tar`
  - use `grep` command to find which file contains the SECRET AUTH CODE
  - write the name of that file to a new file called: `authcode.txt`
  - remove the 1000 files tar created (but leave authcode.txt)[1]
  - print "Found the secret auth code! Saved to authcode.txt" to the screen (not to the `authcode.txt` file)
- Hand in your solution (only `secretfinder.sh`) using AutoLab

[1] Again **be careful** with `rm` and see tips on next page

Some tips

- Begin your file (the very 1st line) with `#!/bin/bash`
  - after that you can comment each line using the # symbol
  - put your name and the date in a comment at the top
  - comment subsequent commands
- To extract files use:

  `tar xf grep.tar`
- To delete the 1000 files, use `rm f*==`

  notice how the above command should only delete files that start with "f" and end with "=="

Scoring

- Total 5 points
- 1 point: file is named properly, begins with interpreter line, valid comments for each line
- 1 point: created directory and extracts files
- 1 point: finds AUTH CODE
- 1 point: writes AUTH CODE to correct file
- 1 point: cleans up 1000 files and prints only one line of output.